TEACHING CLASSICAL CONCURRENT ALGORITHMS USING A GRAPHICAL INTERFACE TOOLKIT - *SesTools*

L. ASSUNÇÃO, N. OLIVEIRA, M. BARATA
Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores. Instituto Superior de Engenharia de Lisboa. Portugal.

*The study of classical concurrent algorithms like Readers/Writers is a key concept on every Operating Systems subject. In this paper we present a graphical tool (**SesTools**) developed on top of the Windows Operating System in order to help students to understand better these algorithms. We also present the conclusions that we have achieved in the scope of the Operating Systems of the Informatics and Computers Engineering curricula at Instituto Superior de Engenharia de Lisboa (ISEL).*

1. Introduction

Multitasking environments and concurrency-control problems such as Readers/Writers and other classical algorithms are key concepts addressed by any introductory Operating Systems subject. However, the study and experimentation based on text console interfaces does not allow students to understand intuitively the behaviour of these algorithms as well as some intrinsic phenomena like starvation and deadlocks.

The context of our teaching experience occurs on first Operating Systems subject at end of the 2nd year (4th semester) of Informatics and Computer Engineering degree at ISEL (Instituto Superior de Engenharia de Lisboa).

Previously students have learned programming techniques, including Object Oriented concepts and programming skills in C/C++ and Java.

We used Microsoft Windows family operating systems to introduce multitasking concepts, e.g., process, thread and the needs to synchronize/control resource-sharing access with mutual exclusion.

In this context we developed some windows based tools with graphical interface to support teaching and students laboratory work.

We defined some C++ wrapper classes such as the Semaphore to reuse better the classic concurrent algorithms presented on reference books about Operating Systems concepts like for instance the Readers-Writers problem [2]. This way we encapsulated the complexity of the thread synchronization objects in the Windows Win32 API. Our pedagogical experience evidenced that using these kinds of tools allowed to achieve a bigger students' motivation and, above all, better results related with algorithms understanding and the capacity to build variants of these algorithms to solve similar problems within new scenarios.

This paper presents in section 2 a brief description of the classical Readers/Writers algorithm. In section 3 we present the ***SesTools*** tools that we developed based on Windows Operating System with a pedagogical GUI interface. Finally in section 4 we present the conclusions and outcomes that we have achieved during last teaching years' experience.

2. The Algorithm – Readers/Writers

The algorithm here presented is based on a basic synchronization object called semaphore. The semantics of a semaphore was defined by EW Dijkstra [1] and can be described as followed: A semaphore *S* is a synchronization object with an integer value *C* non negative (always >=0) which represents the number of semaphore units. The initial value of *C* depends of the utilization scenario of the semaphore, e.g., it has a value of 1 for mutual exclusion and a value of 0 for synchronization between two threads.

A semaphore allows two atomic operations designed **Wait()** and **Signal()** with the semantics presented in Table 1:

```
Wait( ) {
  if (C > 0) // semaphore without units
     C=C-1; // decrement one unit
  else Block_Thread_in_Queue;
}
```

```
Signal( ) {
  if (are_Threads_Blocked_in_Queue)
     Unblock_one_Thread_from_Queue;
  else C=C+1; // increment one unit
}
```

Table 1. Semaphore operations

We implemented a wrapper class encapsulating the Windows operating system Win32 API calls related with semaphore Windows object. After that, the use of semaphore is very simple, it is only necessary to declare and instantiate one object and setup its initial value:

Semaphore *WritersSemaphore = new Semaphore(1).

Readers/Writers Problem

The Readers/Writers is a concurrency-control problem related with the access to a resource, for instance, a database or a simple file shared by multiple concurrent threads [2]. Some of these threads (Readers) may only want to read the resource, whereas others (Writers) may want to write the resource as Figure 1 illustrates.

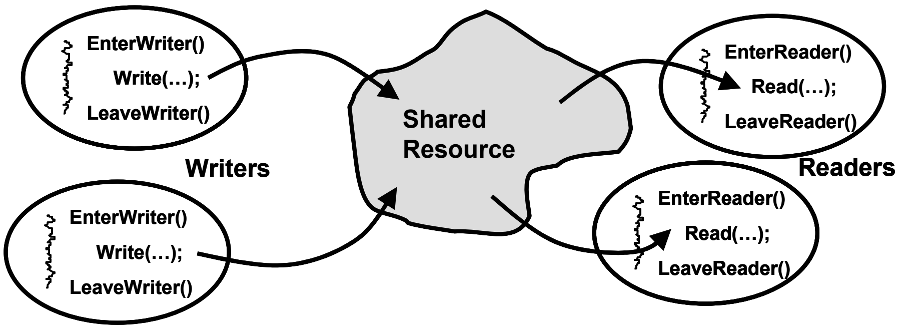


Figure 1. Readers/Writers problem

If two reader threads access the resource simultaneously, no adverse affects will result. However, if a writer thread and some other threads (either a reader or writer) access the resource simultaneously, resource inconsistence might arise. To ensure that these difficulties do not happen, it is necessary the fulfillment of the following requirements:

1) A writer must win exclusive access to the shared resource. When one writer wins the competition to access the resource, other competitors (readers or writers) will wait until they can win the access to the resource;
2) When a reader is accessing the shared resource, there isn't any restriction for another reader to access concurrently the same resource. This requirement will achieve a better performance, when the most frequent pattern corresponds to the number of readings that is larger than writings.

The requirement 1) between writers can be implemented using a synchronization semaphore with an initial value equal to 1 (first writer wins access to the resource).

The requirement 2) between readers can be implemented using a counter of readers and a synchronization semaphore to control the access in mutual exclusion to the counter of readers. This semaphore has an initial value equal to 1 (first reader wins access to the counter).

The classical algorithm to solve the readers/writers problem can be expressed using a C++ class presented in Table 2. The initialization code and the code fragment for a reader and a writer threads are also included.

```
class RdWrProblem {
private:
  int readCount;
  Semaphore *mutexCount;
  Semaphore * WritersSemaphore;
public:
  RdWrProblem () {
    readCount  = 0;
    mutexCount=new Semaphore(1, 1);
    WritersSemaphore = new Semaphore(1,1);
  }
  ~ RdWrProblem () {
      delete mutexCount;
      delete WritersSemaphore;
   }
  void EnterWriter() {
     WritersSemaphore ->Wait();
  }
  void LeaveWriter() {
    WritersSemaphore ->Signal();
  }

  void EnterReader()   {
     mutexCount->Wait();
      if ( ++readCount == 1 )
          WritersSemaphore ->Wait();
     mutexCount->Signal();
   }
   void LeaveReader()   {
     mutexCount->Wait();
      if ( --readCount == 0 )
         WritersSemaphore ->Signal();
     mutexCount->Signal();
   }
};
```

```
// initialization code

RdWrProblem *rsws = new RdWrProblem();

// Reader Thread code fragment
. . .
rsws-> EnterReader();
  // Can Read shared resource
rsws->LeaveReader();
. . .


// Writer Thread code fragment
. . .
rsws-> EnterWriterr();
  // Can Write shared resource
rsws->LeaveWriter();
. . .
```

Table 2. A C++ class to implement the readers/writers problem

Despite the simplicity of this solution, found in all classical Operating Systems books [2], there are some open issues, e.g., which threads have the priority to win the access to the resource? What happens if readers are obsessively reading the resource?

The student's goal, after studying this problem, is to understand why the readers have priority in this solution and if readers are obsessive, the starvation phenomenon might occur.

To achieve the answers to the previous questions is not simple if we only analyze the code. In section 4 we try to explain how we have found better results when we use ***SesTools*** for student's experimentation and achieve the answers to the previous questions.

3. SesTools

The Reader/Writer toolkit of ***SesTools*** is composed by a graphical user interface, a group of reader threads a group of writer threads and a manager object with one implementation of the readers/writers problem (Figure 2). The graphical user interface was developed in C/C++ and is based on the Win32 API [3][4][5]. The readers and writers threads implement a generic life cycle to test the implementation of concurrency control and access control to the shared resource.

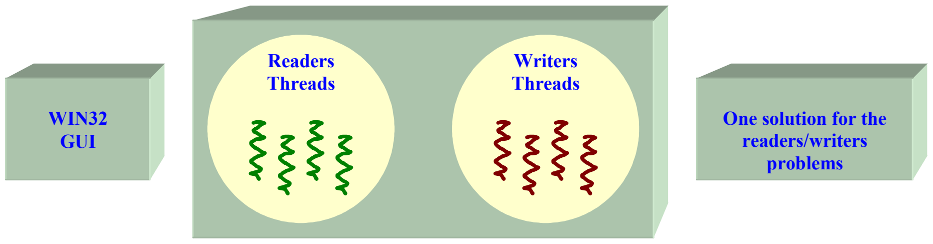


Figure 2. ***SesTools*** – Readers/Writers toolkit Architecture

The threads' code depends on a C++ object of type *ReadersWritersAcess* (Table 3) which defines the interface of the concurrent-control algorithm. As we have seen in the previous section (Table 2) this object has an *EnterReader* method used by readers when they want to win access to the shared resource in reading mode. When a reader finishes the read action it calls the *LeaveReader* method to relinquish its access. Similarly the writers also have their methods: *EnterWriter* and *LeaveWriter*.

The definition of a new solution for the problem implies to create a new class that extends from the abstract class ReadersWritersAcess of Table 3 redefining the four virtual methods. In addition, we must instantiate an object of this new class in the initialization code, compile and run the new version of toolkit. In this way, students may study, implement and test several solutions for the problem. Furthermore, they may define their solution using various WIN32 synchronization mechanisms such as *critical-section*, *mutex*, *semaphore*, etc. [5].

```
class ReadersWritersAcess {
public:
   virtual void EnterReader() = 0;
   virtual void LeaveReader() = 0;
   virtual void EnterWriter() = 0;
   virtual void LeaveWriter() = 0;
};
```

Table 3. Abstract class `ReadersWritersAcess` to define the solution interface

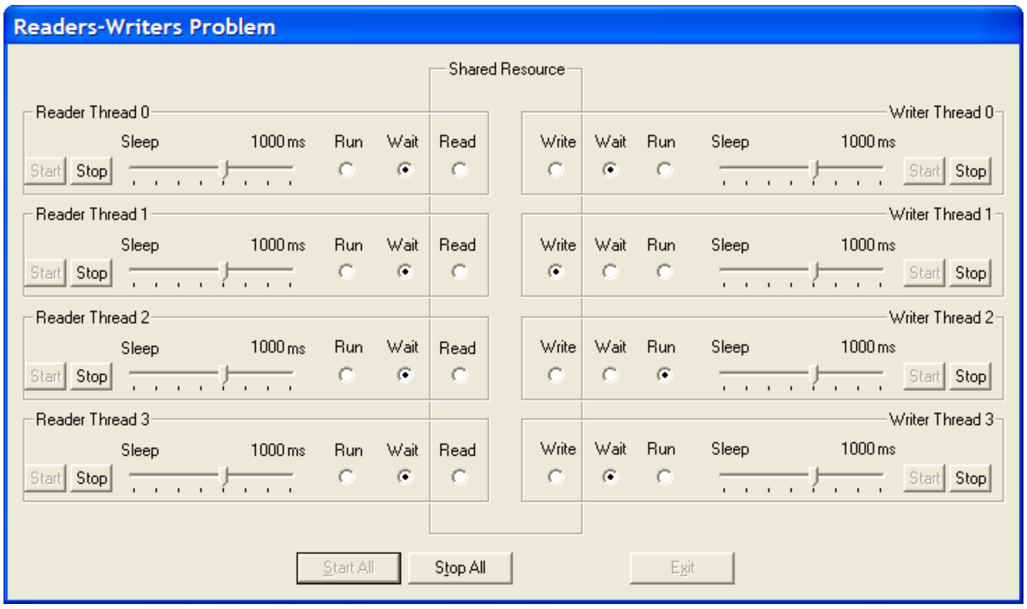
The Readers/Writers toolkit has the graphical user interface (GUI) presented in Figure 3. These GUI permits to control the execution of readers and writers threads. Threads have a life cycle with three phases: i) it executes without accessing the shared resource; ii) it needs to access the resource for reading or writing and; iii) it waits some time to access the resource depending on the state of the resource. We can run up to four reader threads and four writer threads in concurrent execution. Each thread may be started individually by selecting the corresponding start button. We may also start all readers and writers threads through the Start All button. The time that each thread is executing outside the shared resource may be defined in the GUI through a slide bar control of each thread. This control permits the definition of a time between zero and four thousands of milliseconds.

The toolkit presents the actual state of readers and writers threads. Each one can be in one of these three states:

Run – when executing without needs of accessing resource sharing;

Wait – when a thread is waiting to access resource shared to read or write;

Read or Write – when a thread is accessing the resource in reading or writing mode.

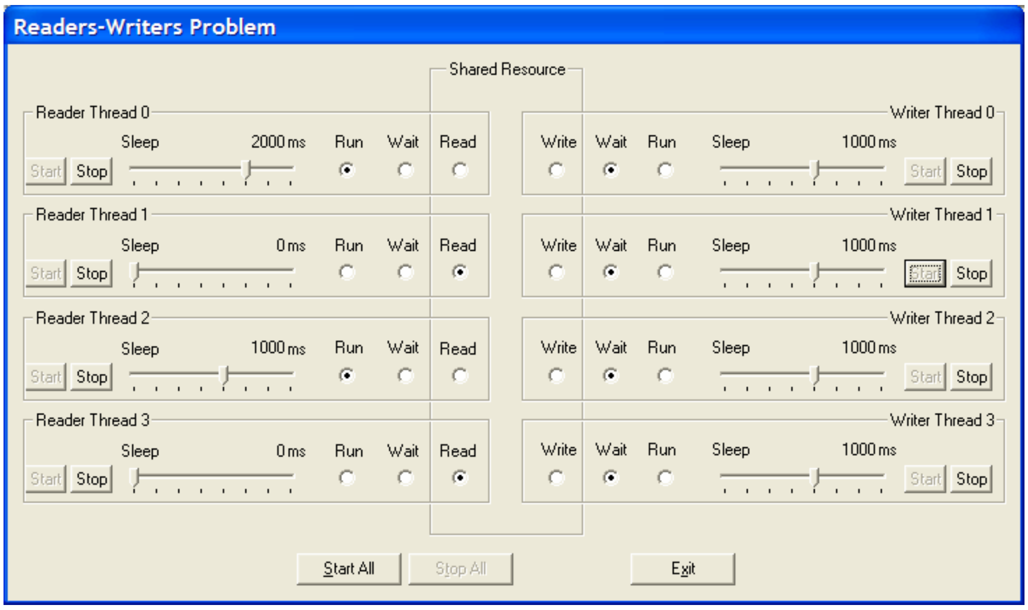


Reader Thread 0 — Sleep 1000 ms — Run [✓ Wait] Read
Reader Thread 1 — Sleep 1000 ms — Run [✓ Wait] Read
Reader Thread 2 — Sleep 1000 ms — Run [✓ Wait] Read
Reader Thread 3 — Sleep 1000 ms — Run [✓ Wait] Read
Writer Thread 0 — Write [✓ Wait] Run — Sleep 1000 ms
Writer Thread 1 — [✓ Write] Wait Run — Sleep 1000 ms
Writer Thread 2 — Write Wait [✓ Run] — Sleep 1000 ms
Writer Thread 3 — Write [✓ Wait] Run — Sleep 1000 ms

Figure 3. GUI of Readers-Writers toolkit

Starvation example

As we can see the solution presented in Table 2 assigns priority to readers so writers may starve. Running this solution on our toolkit we can observe all writer threads waiting to access the shared resource as long as readers are continuing to access the resource in an obsessive way (Figure 4). Consequently the writers don't have the possibility to access the resource.

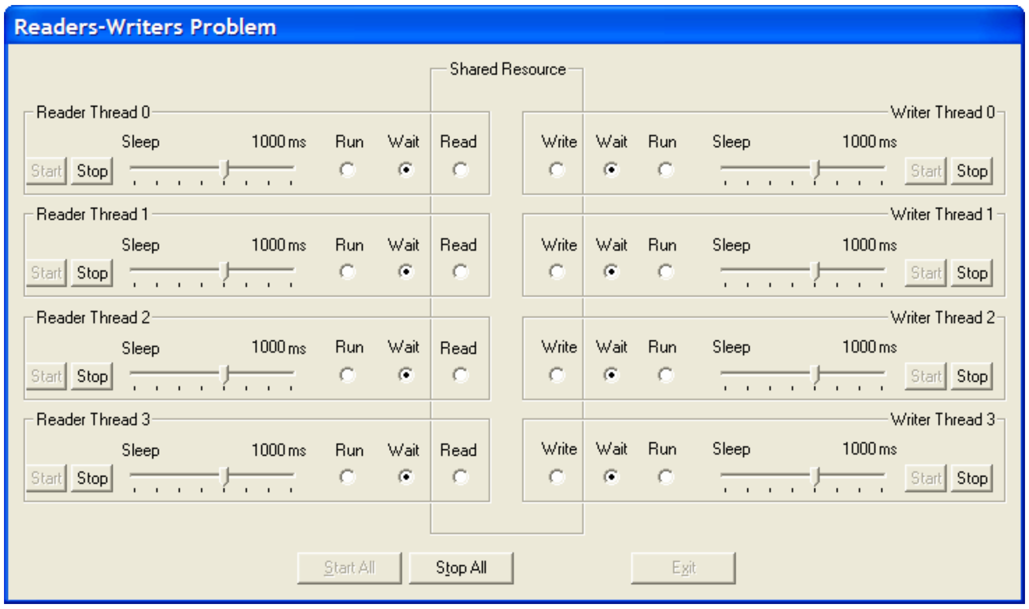


Reader Thread 0 — Sleep 2000 ms — [✓ Run] Wait Read
Reader Thread 1 — Sleep 0 ms — Run Wait [✓ Read]
Reader Thread 2 — Sleep 1000 ms — [✓ Run] Wait Read
Reader Thread 3 — Sleep 0 ms — Run Wait [✓ Read]
Writer Thread 0 — Write [✓ Wait] Run — Sleep 1000 ms
Writer Thread 1 — Write [✓ Wait] Run — Sleep 1000 ms
Writer Thread 2 — Write [✓ Wait] Run — Sleep 1000 ms
Writer Thread 3 — Write [✓ Wait] Run — Sleep 1000 ms

Figure 4. Toolkit running a solution which gives priority to readers and we can see writers threads in starvation

Deadlock example

Sometimes a erroneous solution may cause a deadlock. This situation is very frequent in the first students' solutions. The readers/writers toolkit permits the visualization of the deadlock situation as we can see in Figure 5, where all threads (readers and writers) are in the wait state. Consequently the resource is available but due to an erroneous solution any thread is allowed to access the resource!



Reader Thread 0 Start Stop Sleep 1000 ms Run [✓ Wait] Read
Reader Thread 1 Start Stop Sleep 1000 ms Run [✓ Wait] Read
Reader Thread 2 Start Stop Sleep 1000 ms Run [✓ Wait] Read
Reader Thread 3 Start Stop Sleep 1000 ms Run [✓ Wait] Read
Writer Thread 0 Write [✓ Wait] Run Sleep 1000 ms Start Stop
Writer Thread 1 Write [✓ Wait] Run Sleep 1000 ms Start Stop
Writer Thread 2 Write [✓ Wait] Run Sleep 1000 ms Start Stop
Writer Thread 3 Write [✓ Wait] Run Sleep 1000 ms Start Stop

Figure 5. Readers-Writers toolkit running an erroneous solution that causes a deadlock situation

The toolkit presented in this paper is available for downloading at http://phoenix.deetc.isel.ipl.pt/downloads/SesTools.

4. Conclusions

Before we introduced ***SesTools*** we got less than 50% of correct answers when doing assessments where we asked questions like, “explain what you understand about the starvation phenomenon in a multitask environment” or “for a readers/writers classical algorithm describe the reasons why readers have priority versus writers”.

After we introduced in classroom ***SesTools*** we realized more than 75% of the students were capable to answer correctly the previous questions and mainly they were able to discuss new versions of the algorithm; e. g., how they can change the algorithm so that the writers can have priority versus readers or a fairness solution.

Student’s motivation is another result. We have achieved that students were more motivated and by own initiative they experiment and self-test their solutions. We have also seen the students doing similar graphical user interfaces to new exercises that we have proposed.

Last year our colleagues that taught Operating Systems concepts in Electronics and Telecommunications course used and validated the ideas presented in this paper rewriting a new toolkit based on Java language and the results were also good.

References

[1] E. W. Dijkstra, The Structure of the "THE"-Multiprogramming System, Communications of the ACM, May 1968.
[2] A. Siberschatz, P. Galvin, G. Gagne, Operating System Concepts, Sixth Edition, John Wiley & Sons, 2004.
[3] C. Petzold, Programming Windows, Fifth Edition, Microsoft Press; 5th edition, 11 November 1998.
[4] J. Richter, Programming Applications for Microsoft Windows, Microsoft Press, October 1999.
[5] Microsoft, msdn library, January 2006